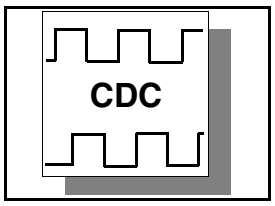
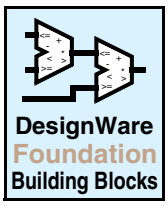




DW_stream_sync

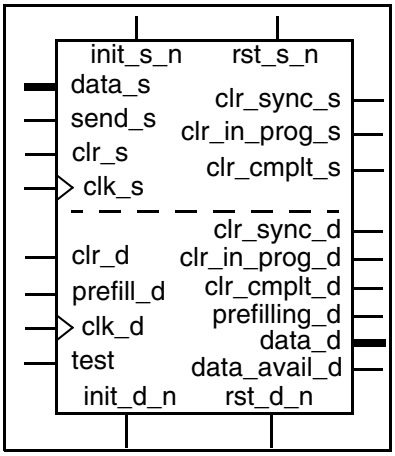
Data Stream Synchronizer

Version, STAR, and myDesignWare Subscriptions: IP Directory

Features and Benefits

- Interface between dual asynchronous clock domains
- Coordinated clearing between clock domains
- Parameterized data bus width
- Parameterized number of synchronizing stages
- Parameterized test feature
- All data-related outputs registered
- Ability to model missampling of data on source clock domain

Revision History



Description

The DW_stream_sync passes a data stream from the source domain to the destination domain with a minimum amount of latency. As long as the aggregate data rate from the source domain does not exceed the destination flow rate, a wide variety of disparate clock rates can be used. Using the parameter *prefill_lvl*, the data stream FIFO can be prefilled to a predetermined level for use as an elasticity buffer. A single `clk_d` cycle pulse of the `prefill_d` input directs the first-in-first-out (FIFO) to be prefilled during which time a destination domain output called `prefilling_d` is asserted when the number of valid entries in the FIFO is below the threshold set by *prefill_lvl*. Full feedback hand-shake is not used, so there is no busy or done status on in the source domain.

A unique built-in verification feature enables you to turn on a random sampling error mechanism that models skew between bits of the incoming data bus from the source domain (for more information, see "Simulation Methodology" on page 8). This facility provides an opportunity for determining system robustness during the early development phases without having to develop special test stimulus.

Note As of DesignWare versions F-2011.09-SP1 and later, the underlying component DW_reset_sync was enhanced to improve the clearing sequence. As a result, Figure 1-3, Figure 1-4 and Figure 1-5 of this datasheet have been updated to reflect the change of behavior of the clearing sequence. This enhancement of DW_reset_sync does not impact the basic function of DW_stream_sync with respect to the streaming of data before, during, and after the clearing sequence. For more details about the changes to DW_reset_sync refer to the DW_reset_sync datasheet.

Table 1-1 Pin Description

Pin Name	Width	Direction	Function
clk_s	1	Input	Source Domain clock source
rst_s_n	1	Input	Source Domain asynchronous reset (active low)
init_s_n	1	Input	Source Domain synchronous reset (active low)
clr_s	1	Input	Source Domain clear
send_s	1	Input	Source initiate valid data vector control
data_s	width	Input	Source Domain data vector
clr_sync_s	1	Output	Source Domain clear for sequential logic
clr_in_prog_s	1	Output	Source Domain clear sequence in progress
clr_cmplt_s	1	Output	Source Domain that clear sequence complete (single `clk_s` cycle pulse)
clk_d	1	Input	Destination Domain clock source
rst_d_n	1	Input	Destination Domain asynchronous reset (active low)
init_d_n	1	Input	Destination Domain synchronous reset (active low)
clr_d	1	Input	Destination Domain clear
prefill_d	1	Input	Destination Domain prefill control
clr_in_prog_d	1	Output	Destination Domain clear sequence in progress
clr_sync_d	1	Output	Destination Domain clear for sequential logic
clr_cmplt_d	1	Output	Destination Domain that clear sequence complete (single `clk_d` cycle pulse)
data_avail_d	1	Output	Destination Domain data update
data_d	width	Output	Destination Domain data vector
prefilling_d	1	Output	Destination Domain prefilling FIFO in progress
test	1	Input	Scan test mode select

Table 1-2 Parameter Description

Parameter	Values	Description
width	1 to 1024 Default: 8	Vector width of input `data_s` and output `data_d`
depth	1 to 256	Depth of FIFO
prefill_lvl	0 to depth-1 Default: 0	The number of valid entries in the FIFO before transferring packets to destination (enabled when `prefill_d` asserted)

Table 1-2 Parameter Description (Continued)

Parameter	Values	Description
f_sync_type	0 to 4 Default: 2	Forward Synchronization Type Defines type and number of synchronizing stages: ■ 0: Single clock design, no synchronizing stages implemented ■ 1: 2-stage synchronization with first stage negative-edge capturing and second stage positive-edge capturing ■ 2: 2-stage synchronization with both stages positive-edge capturing ■ 3: 3-stage synchronization with all stages positive-edge capturing ■ 4: 4-stage synchronization with all stages positive-edge capturing
reg_stat	0 or 1 Default: 1	Registering Internal Status (affects `prefilling_d`) ■ 0: Do not register internally calculated status (`prefilling_d` appears one cycle sooner than if *reg_stat* = 1) ■ 1: Register internally calculated status
tst_mode	0 to 2 Default: 0	Test Mode ■ 0: No latch is inserted for scan testing ■ 1: Insert negative-edge capturing register on `data_s` input vector when test input is asserted ■ 2: Insert hold latch using active low latch
verif_en	0 to 4 Default: 2	Verification Enable Control ■ 0: No sampling errors inserted ■ 1: Sampling errors randomly inserted with 0 or up to 1 destination clock cycle delays ■ 2: Sampling errors randomly inserted with 0, 0.5, 1, or 1.5 destination clock cycle delays ■ 3: Sampling errors randomly inserted with 0, 1, 2, or 3 destination clock cycle delays ■ 4: Sampling errors randomly inserted with 0 or up to 0.5 destination clock cycle delays For more information, see “Simulation Methodology” on page 8.
r_sync_type	0 to 4 Default: 2	Reverse Synchronization Type (Destination to Source Domains) ■ 0: No synchronization, single clock system ■ 1: 2-stage synchronization w/ 1st stage negative-edge and 2nd stage positive-edge capturing ■ 2: 2-stage synchronization w/ both stages positive-edge capturing ■ 3: 3-stage synchronization w/ all stages positive-edge capturing ■ 4: 4-stage synchronization w/ all stages positive-edge capturing
clk_d_faster	0 to 15 Default: 1	Obsolete parameter; the value setting is ignored This parameter is kept in place only for backward compatibility.

Table 1-2 Parameter Description (Continued)

Parameter	Values	Description
reg_in_prog	0 or 1 Default: 1	Register the `clr_in_prog_s` and `clr_in_prog_d` Outputs ■ 0: Unregistered ■ 1: Registered

Table 1-3 Synthesis Implementations

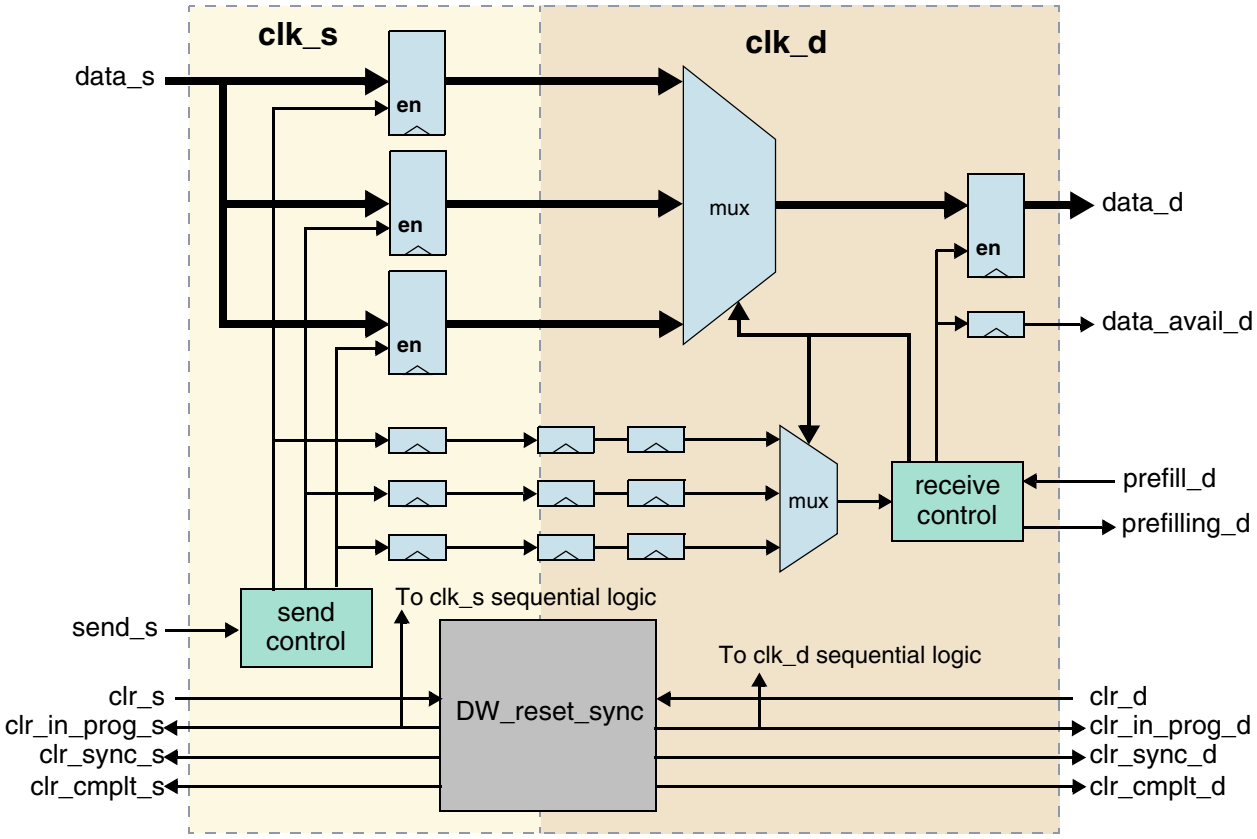
Implementation Name	Function	License Feature Required
rtl	Synthesis model	DesignWare

Table 1-4 Simulation Models

Model	Function
DW03.DW_STREAM_SYNC_CFG_SIM	Design unit name for VHDL simulation
DW03.DW_STREAM_SYNC_CFG_SIM_MS	Design unit name for VHDL simulation with mis-sampling enabled
dw/dw03/src/DW_stream_sync_sim.vhd	VHDL simulation model source code (modeling RTL)—with or without missampling For usage, see "Simulation Methodology" below.
dw/sim_ver/DW_stream_sync.v	Verilog simulation model source code

Block Diagram

Figure 1-1 DW_stream_sync Basic Block Diagram



Reset Considerations

System Resets (synchronous and asynchronous)

The instantiated DW_reset_sync converts the clearing inputs (`clr_s` and `clr_d`) to 'toggle' events (from the source and destination domains, respectively). These 'toggle' events do not rely on state value but rather on state change from the originating domain. As a result, an assertion of `rst_s_n` or `init_s_n` could cause an erroneous 'toggle' event to occur that translates over into the destination domain in a 'false' assertion of `clr_sync_d`. Similarly, an assertion of `rst_d_n` or `init_d_n`, could result in a 'false' assertion of `clr_sync_s`. This inherently could cause inconsistencies when the activation of these resets is not coordinated between the two domains at the system level. Therefore, as a requirement to insure clean set and release of the reset state, both source and destination domains must be, at some point, in the active reset state simultaneously when performing system resets. That is, when asserting `rst_s_n` and/or `init_s_n`, then `rst_d_n` and/or `init_d_n` must also be asserted at the same time and vice versa. As a general requirement the length of the system reset signal(s) assertion should be a minimum of 4 clock cycles of the slowest clock between the two domains. System reset signals for both clock domains, when asserted, should overlap for a minimum of *f_sync_type* + 1 or *r_sync_type* + 1 cycles (which ever is larger) of the slowest clock of the two domains.

Besides satisfying simultaneous assertion of each domains system reset signals for a minimum number of cycles, the timing of the assertion between these signals needs some consideration. To prevent erroneous `clr_sync_s` and `clr_sync_d` pulses from occurring when system resets are asserted, it is recommended that:

1. If the source domain reset is asserted first, then the destination domain should assert its reset within *f_sync_type* + 1 `clk_d` cycles from the time the assertion of the source domain reset occurred OR
2. If the destination domain reset is asserted first, then the source domain should assert its within *r_sync_type* + 1 `clk_s` cycles from the time the assertion of the destination domain reset occurred.
3. If both domains can tolerate a false `clr_sync_s` or `clr_sync_d` (whichever the case) during system reset conditions, then this recommendation can be ignored as long as both clock domains eventually have overlapping active reset conditions.

There are no restrictions on when to release the reset condition on either side. However, to be completely safe, it is recommended, though not required, to release the source clock domain's reset last.

Additionally, the clearing signals (`clr_s` and `clr_d`) should not be asserted sooner than one clock cycle after their respective domain's system reset de-assertion. In fact, if possible, the first assertion of `clr_s` or `clr_d` shouldn't occur until one clock cycle within its domain from the last de-assertion of system reset between both domains.

Local Clearing (synchronous clearing)

The DW_stream_sync contains a DW_reset_sync module that provides clearing signals for each domain, called 'clr_s' and 'clr_d'. A minimum of a single clock cycle pulse on either one of these clearing signals initiates a synchronized clearing sequence to each domain for resetting of sequential elements of the system. This clearing sequence is orchestrated to ensure that the destination domain interface is completely cleared and ready before the source domain is permitted to initiating new activity.

In general, independent of which domain initiates the clearing sequence, the destination domain always goes into the active clearing state before the source domain does as indicated by `clr_in_prog_d` and

clr_in_prog_s going to '1', respectively. Similarly, the destination domain exits the active clearing state before the source domain does as indicated by clr_in_prog_d and clr_in_prog_s going to '0', respectively.

Figure 1-3 on page 11 through Figure 1-5 on page 13 show various timing for clr_s and clr_d.

It is imperative for system integrity to cease source domain activity after asserting clr_s (and while waiting for a subsequent clr_cmplt_s pulse) and/or when observing an active clr_in_prog_s. From the destination domain, accepting activity after clr_d is asserted and/or observing an active clr_in_prog_d would result in corrupting system integrity. Ultimately, it is important to halt source domain and destination domain activity during the clearing sequence and only start source domain activity after the clr_cmplt_s pulse is observed.

There is no restriction on how often or how long clr_s and clr_d can be asserted. The clearing operation is maintained if in progress and subsequent clr_s and/or clr_d initiations are made. Once the final assertion of clr_s and/or clr_d is made, the sustained clearing sequence eventually comes to completion and all 'in progress' flags de-assert.

Test

The synthesis parameter, *tst_mode*, controls the insertion of lock-up latches at the points where signals cross between the clock domains, clk_s and clk_d. Lock-up latches are used to ensure proper cross-domain operation during the capture phase of scan testing in devices with multiple clocks. When *tst_mode* = 1, lock-up latches will be inserted during synthesis and will be controlled by the input, test.

With *tst_mode* = 1, the input, test, controls the bypass of the latches for normal operation where test=0 bypasses latches and test = 1 includes latches. In order to assist DFT compiler in the use of the lock-up latches, use the set_test_hold 1 tst_mode command before using the insert_scan command.

When *tst_mode* = 0 (which is its default value when not set in the design) no lock-up latches are inserted and the test input is not connected.

Note: The insertion of lock-up latches requires the availability of an active low enable latch cell. If the target library does not have such a latch or if latches are not allowed (using dont_use commands for instance), synthesis of this module with *tst_mode* = 1 will fail.

Simulation Methodology

Because this component contains synchronizing devices, there are two methods available for simulation. One method is to use the simulation models that emulate the RTL model. Or, you can enable modeling of random skew between bits of signals traversing to and from each domain (called missampling).

To use the simulation models that emulate the RTL model, no special configuration is required.

To use missampling requires the following considerations:

- To enable missampling in Verilog simulations, define the macro DW_MODEL_MISSAMPLES:

  ```
  `define DW_MODEL_MISSAMPLES
  ```

  If `DW_MODEL_MISSAMPLES is defined, the *verif_en* parameter comes into play to configure the simulation model as described by Table 1-2 on page 2. If `DW_MODEL_MISSAMPLES is not defined, the Verilog simulation model behaves as if *verif_en* is set to 0.

- To enable missampling in VHDL simulations, a simulation architecture named sim_ms is provided. The parameter *verif_en* only has meaning when using sim_ms. That is, when the `sim` simulation architecture is used instead, the model behaves as though *verif_en* is set to 0. For examples of how each architecture is used, see "HDL Usage Through Component Instantiation - VHDL" on page 14.

Suppressing Warning Messages During Verilog Simulation

The Verilog simulation model includes macros that allow you to suppress warning messages during simulation.

To suppress all warning messages for all DWBB components, define the DW_SUPPRESS_WARN macro in either of the following ways:

- Specify the Verilog preprocessing macro in Verilog code:

  ```
  `define DW_SUPPRESS_WARN
  ```

- Or, include a command line option to the simulator, such as:

  `+define+DW_SUPPRESS_WARN` (which is used for the Synopsys VCS simulator)

The warning messages for this model include the following:

- If values other than 1 or 0 are present on a clock port, the following message is displayed:

  ```
  WARNING: <instance_path>.<clock_name>_monitor:
      at time = <timestamp>, Detected unknown value, x, on <clock_name> input.
  ```

  To suppress only this warning message for all DWBB components, use the following macro:

  - Define the DW_DISABLE_CLK_MONITOR macro. You can define this macro in the following ways:
    - Specify the Verilog preprocessing macro in Verilog code:

      ```
      `define DW_DISABLE_CLK_MONITOR
      ```

    - Or, include a command line option to the simulator, such as:

      `+define+DW_DISABLE_CLK_MONITOR` (which is used for the Synopsys VCS simulator)

  This message is also suppressed using the DW_SUPPRESS_WARN macro explained earlier.

Timing Diagrams

Figure 1-2 on page 9 depicts the case in which the destination domain clock (`clk_d`) is slightly faster than the source domain clock (`clk_s`) with a contiguous data burst and the assertion of prefill_d. To maintain a contiguous packet with no bubbles received at the destination domain, the FIFO is instructed to fill up with two valid entries (`prefill_lvl` is 2) from the source domain before transferring packets. If the `prefill_d` was not asserted (to allow prefilling of the FIFO) and with `clk_d` faster than `clk_s`, there could be cycles of `clk_d` where `data_avail_d` would be de-asserted for a packet that was meant to be contiguous. In this example, *depth* is 6, width is 8, *f_sync_type* is 2, *prefill_lvl* is 2, *reg_stat* is 1, *tst_mode* is 0, *verif_en* is 0, *r_sync_type* is 2, and *reg_in_prog* is 1. The signal at the bottom, `count_d[2:0]` represents the synchronized (to the destination domain) number of valid entries in the FIFO and is shown for reference only.

Figure 1-2 clk_d Faster than clk_s and Assertion of prefill_d

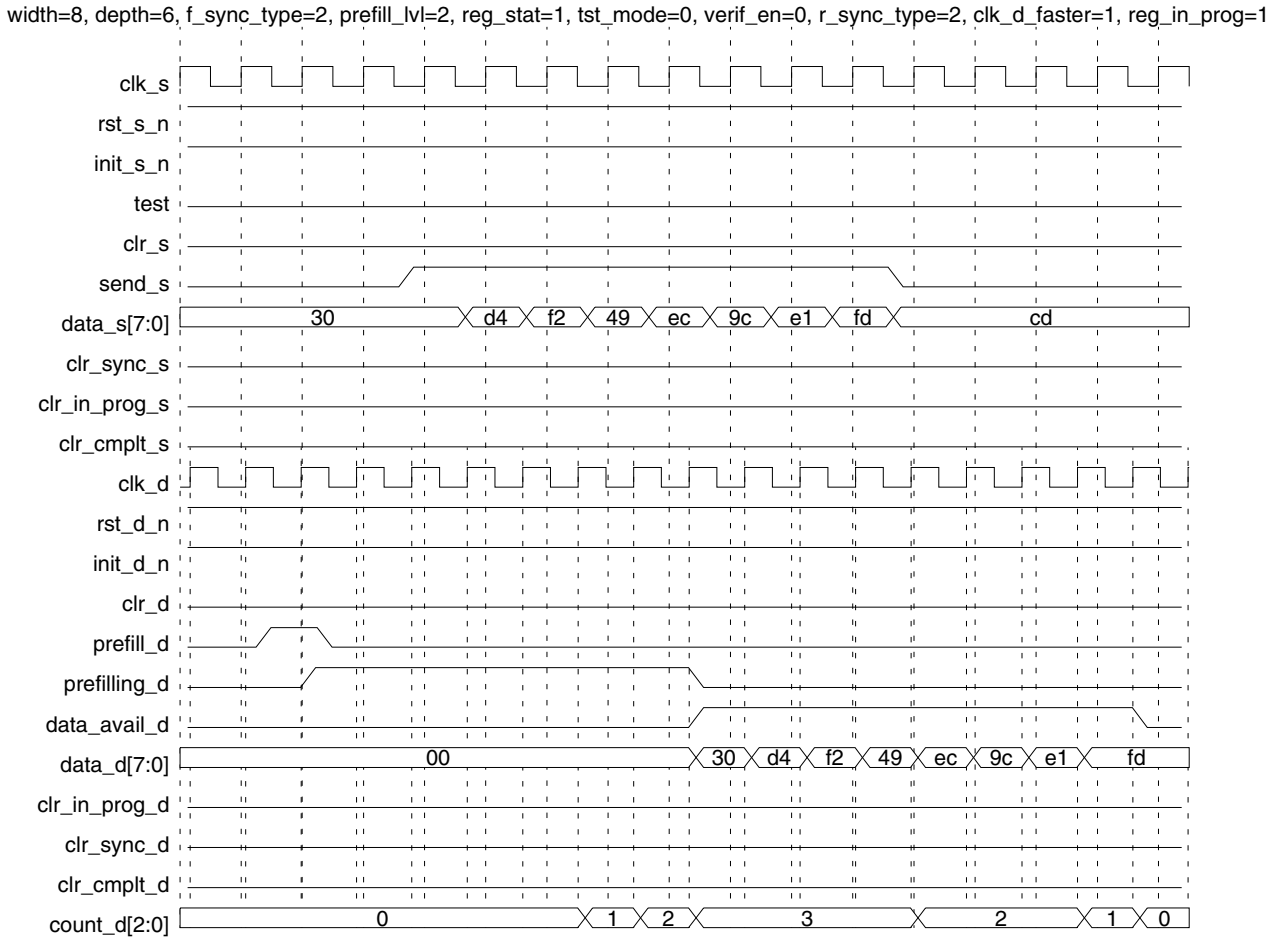


Figure 1-3 on page 11 shows an example of clr_s initiating a local clearing sequence of the DW_stream_sync. When clr_s is asserted, it propagates to the destination domain where it is synchronized and produces the assertion of clr_in_prog_d (which stays active until the clearing sequence from the destination domain's perspective completes). The event of the clr_in_prog_d assertion is fed back to the source domain where it is synchronized and, in turn, starts the source domain clearing event in the form of clr_sync_s and clr_in_prog_s assertions. As with the clr_in_prog_d for the destination domain, clr_in_prog_s stays asserted until the completion of the clearing sequence in the source domain.

The event of the clr_sync_s assertion then gets routed back to the destination domain to initiate the synchronized completion of the clearing sequence on that end. This is indicated by the clr_sync_d pulse and de-assertion of clr_in_prog_d followed by the clr_cmplt_d. At this point, the destination domain is in its initialized state and ready to receive data streams from the source domain.

To finish up the clearing sequence between the two domains, the clr_sync_d pulse is sent back to the source domain where it is synchronized and de-asserts clr_in_prog_s which is followed by the clr_cmplt_s pulse. Once the clr_cmplt_s gets asserted, it is the indication to the source domain that the destination domain is cleared and ready for the new data streams.

Internally, the clr_in_prog_s and clr_in_prog_d are used to reset their respective domain's sequential elements; this is evident as seen for dated as it goes to '0x00' on the next rising-edge of clk_d after the assertion of clr_in_prog_d.

For this waveform example *f_sync_type* is 2, *r_sync_type* is 2, and *reg_in_prog* is 1 (relevant parameters with regard to the clearing sequence).

Figure 1-3 Example of clr_s Initiated Clearing Sequence

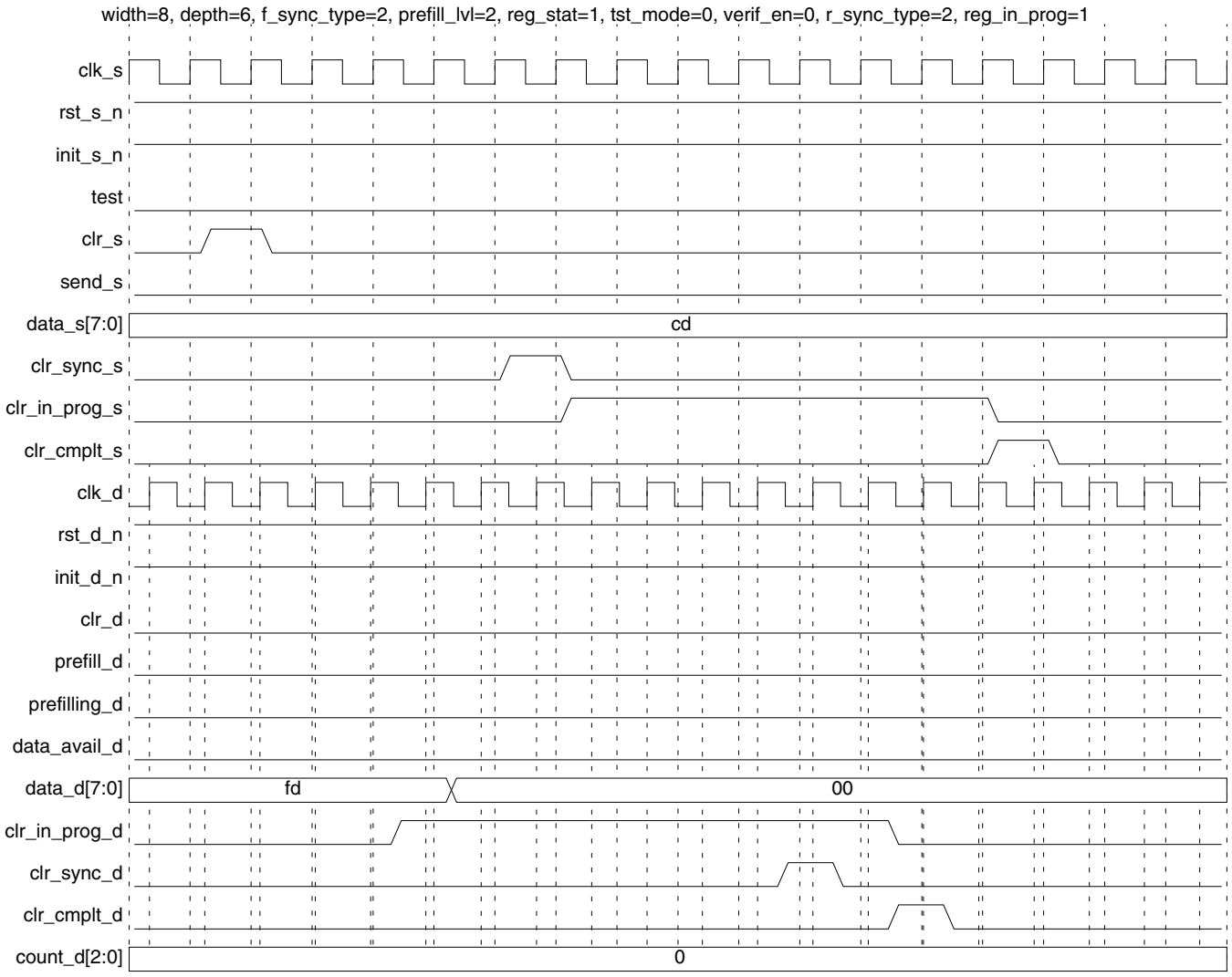


Figure 1-4 on page 12 shows an example of `clr_d` initiating a clearing sequence. The `clr_d` initiated pulse places the destination domain in the clearing state as indicated by the assertion of `clr_in_prog_d`. At the same time, `clr_d` gets sent to the source domain where it is synchronized and triggers the `clr_sync_s` and `clr_in_prog_s` signals. At this point, the source domain needs to realize, based on `clr_sync_s` or `clr_in_prog_s` assertions that the destination domain is in the clearing state and is not receiving any more data.

Once the `clr_sync_s` assertion occurs it is re-synchronized back in the destination domain to generate the `clr_sync_d` output as well as the de-assertion of the `clr_in_prog_d` output followed by the active pulse of `clr_cmplt_d`. At this point, the destination domain is in its initialized state and ready to receive data streams from the source domain.

The assertion of `clr_sync_d` is sent back to the source domain which triggers the de-assertion of `clr_in_prog_s` followed by the `clr_cmplt_s` pulse activation. Once the `clr_cmplt_s` gets asserted, it

is the indication to the source domain that the destination domain is cleared and ready for the new data streams.

Internally, the `clr_in_prog_s` and `clr_in_prog_d` are used to reset their respective domain sequential elements; this is most evident as seen when `data_d` goes to '0x00' and `data_avail_d` goes to '0' on the next rising-edge of `clk_d` after the assertion of `clr_in_prog_d` even though there is a valid data packet of '0x40' on the heals of '0x50' as sent from the source domain.

Parameter settings: *width* = 8, *depth* = 6, *prefill_lvl* = 2, *f_sync_type* = 2, *reg_stat* = 1, *tst_mode* = 1, *verif_en* = 0, *r_sync_type* = 2, and *reg_in_prog* = 1.

Figure 1-4 Example of clr_d Initiated Clearing Sequence

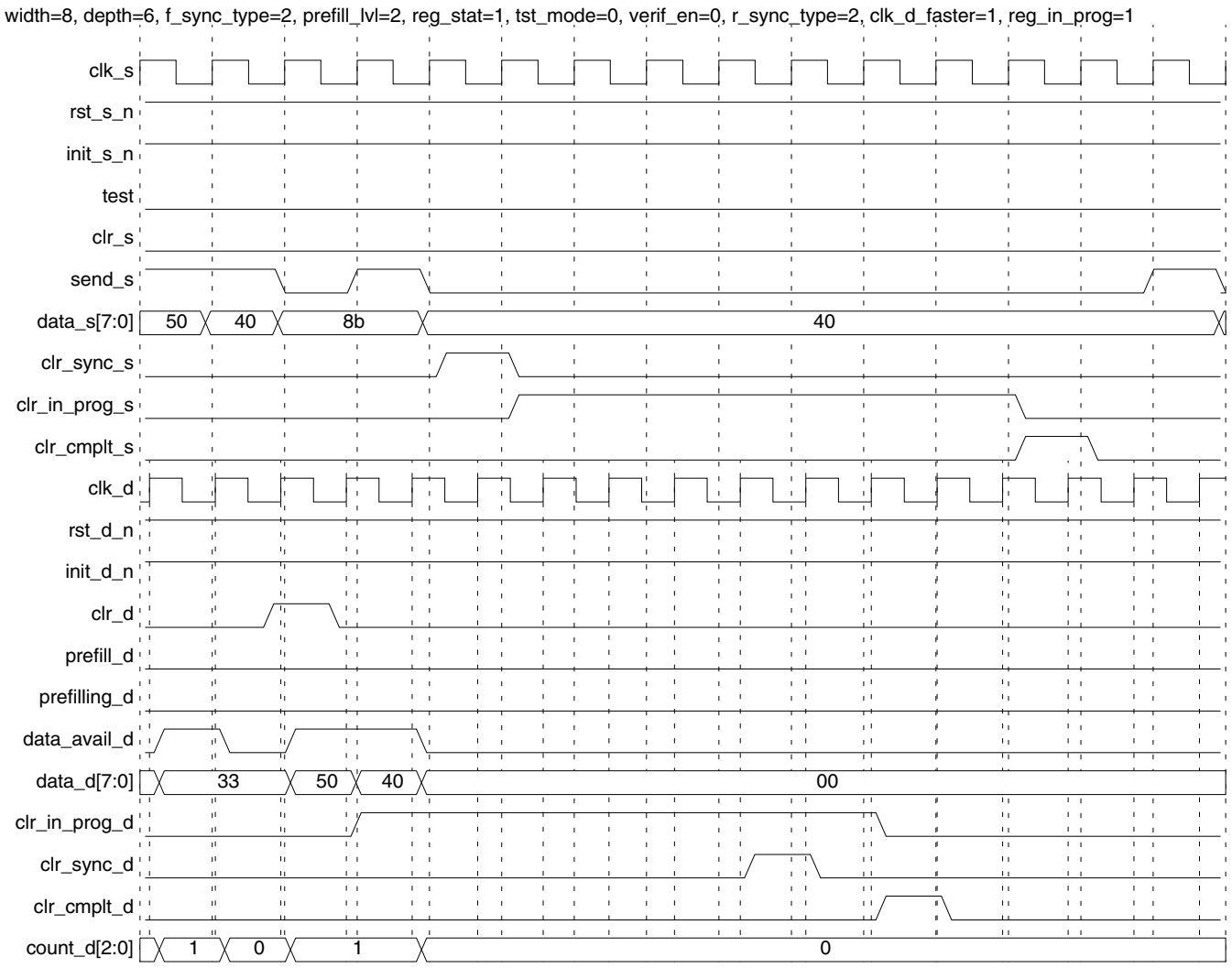
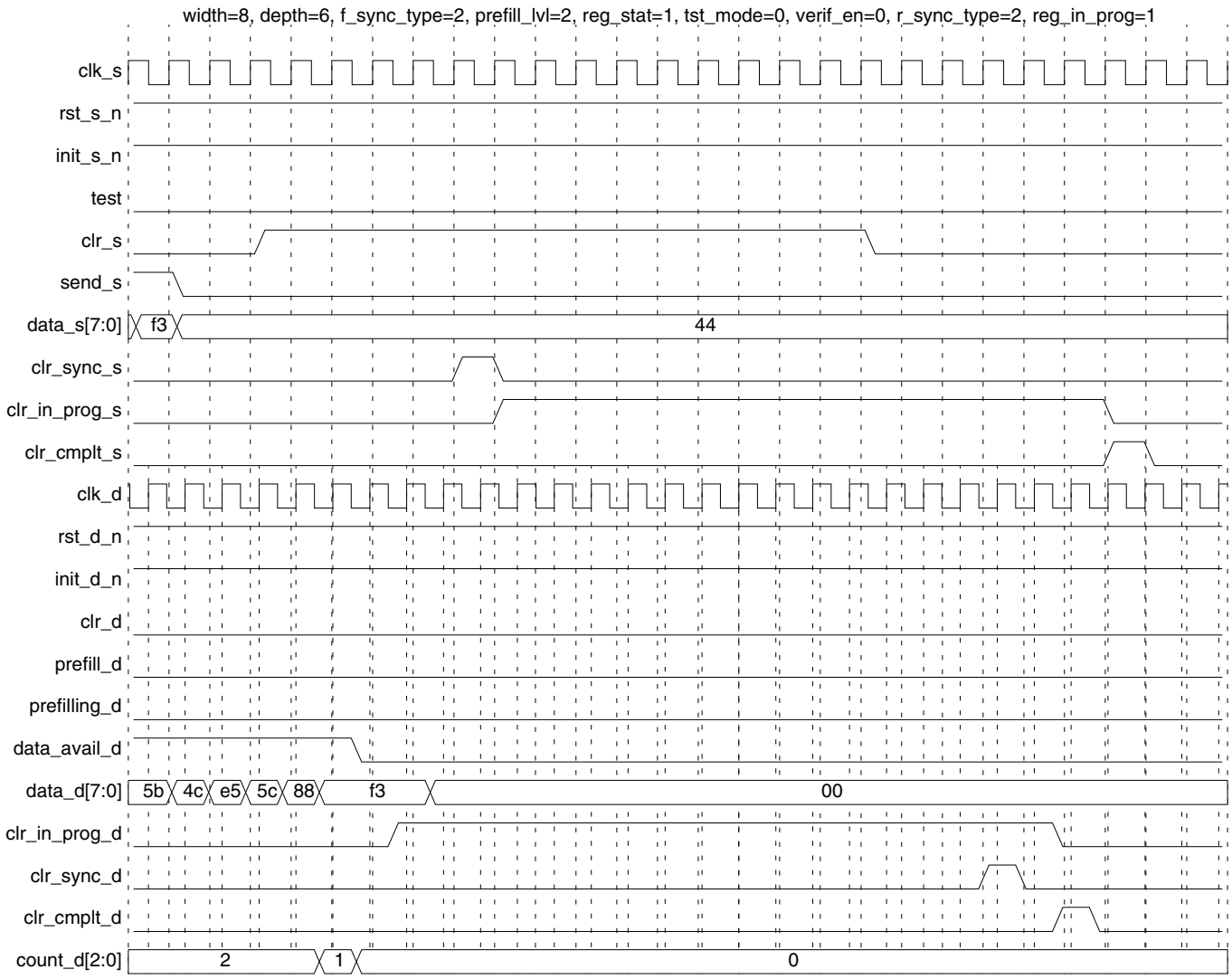


Figure 1-5 shows an initiation of a `clr_s` where its duration is much longer than one `clk_s` cycle. From this, the behavior of the `clr_in_prog_s` and `clr_in_prog_d` flags are sustained longer than those seen in Figure 1-3 on page 11 in which `clr_s` was only asserted a single `clk_s` cycle.

Figure 1-5 Example of sustained clr_s initiated clearing sequence.



Related Topics

- Memory – Registers Overview
- DesignWare Building Block IP User Guide

HDL Usage Through Component Instantiation - VHDL

```
library IEEE,DWARE;
use IEEE.std_logic_1164.all;
use DWARE.DW_Foundation_comp.all;

entity DW_stream_sync_inst is
      generic (
        inst_width : INTEGER := 8;
        inst_depth : INTEGER := 4;
        inst_prefill_lvl : INTEGER := 0;
        inst_f_sync_type : INTEGER := 2;
        inst_reg_stat : INTEGER := 1;
        inst_tst_mode : INTEGER := 0;
        inst_verif_en : INTEGER := 2;
            inst_r_sync_type  : NATURAL  := 2;
            inst_clk_d_faster : NATURAL  range 0 to 15   := 1;
            inst_reg_in_prog  : NATURAL  range 0 to 1    := 1
        );
      port (
        inst_clk_s : in std_logic;
        inst_rst_s_n : in std_logic;
        inst_init_s_n : in std_logic;
        inst_clr_s : in std_logic;
        inst_send_s : in std_logic;
        inst_data_s : in std_logic_vector(inst_width-1 downto 0);
        clr_sync_s_inst : out std_logic;
        clr_in_prog_s_inst : out std_logic;
        clr_cmplt_s_inst : out std_logic;

        inst_clk_d : in std_logic;
        inst_rst_d_n : in std_logic;
        inst_init_d_n : in std_logic;
        inst_clr_d : in std_logic;
        inst_prefill_d : in std_logic;
            clr_in_prog_d_inst : out std_logic;
        clr_sync_d_inst : out std_logic;
            clr_cmplt_d_inst   : out std_logic;
        data_avail_d_inst : out std_logic;
        data_d_inst : out std_logic_vector(inst_width-1 downto 0);
        prefilling_d_inst : out std_logic;

        inst_test : in std_logic
        );
    end DW_stream_sync_inst;


architecture inst of DW_stream_sync_inst is
begin
```

```
    -- Instance of DW_stream_sync
    U1 : DW_stream_sync
    generic map ( width => inst_width, depth => inst_depth, prefill_lvl =>
inst_prefill_lvl,
                f_sync_type => inst_f_sync_type, reg_stat => inst_reg_stat,
                tst_mode => inst_tst_mode, verif_en => inst_verif_en,
          r_sync_type => inst_r_sync_type, clk_d_faster => inst_clk_d_faster,
                     reg_in_prog => inst_reg_in_prog )
port map ( clk_s => inst_clk_s, rst_s_n => inst_rst_s_n, init_s_n =>
                     inst_init_s_n,
    clr_s => inst_clr_s, send_s => inst_send_s, data_s => inst_data_s,
     clr_sync_s => clr_sync_s_inst, clr_in_prog_s => clr_in_prog_s_inst,
     clr_cmplt_s => clr_cmplt_s_inst,
     clk_d => inst_clk_d, rst_d_n => inst_rst_d_n, init_d_n => inst_init_d_n,
     clr_d => inst_clr_d, prefill_d => inst_prefill_d,
     clr_in_prog_d => clr_in_prog_d_inst, clr_sync_d => clr_sync_d_inst,
     clr_cmplt_d => clr_cmplt_d_inst, data_avail_d => data_avail_d_inst,
     data_d => data_d_inst, prefilling_d => prefilling_d_inst, test =>
                          inst_test );


end inst;

-- Configuration for use with a VHDL simulator
-- pragma translate_off
library DW03;
configuration DW_stream_sync_inst_cfg_inst of DW_stream_sync_inst is
  for inst
    -- NOTE: If desiring to model missampling, uncomment the following
    -- line.  Doing so, however, will cause inconsequential errors
    -- when analyzing or reading this configuration before synthesis.
    -- for U1 : DW_stream_sync use configuration DW03.DW_stream_sync_cfg_sim_ms;  end
for;
  end for; -- inst
end DW_stream_sync_inst_cfg_inst;
-- pragma translate_on
```

HDL Usage Through Component Instantiation - Verilog

```
module DW_stream_sync_inst( inst_clk_s, inst_rst_s_n, inst_init_s_n,
    inst_clr_s, inst_send_s, inst_data_s,
    clr_sync_s_inst, clr_in_prog_s_inst, clr_cmplt_s_inst,
    inst_clk_d, inst_rst_d_n, inst_init_d_n, inst_clr_d, inst_prefill_d,
    clr_in_prog_d_inst, clr_sync_d_inst, clr_cmplt_d_inst,
    data_avail_d_inst, data_d_inst, prefilling_d_inst, inst_test );

parameter width        = 8;  // RANGE 1 to 1024
parameter depth        = 4;  // RANGE 2 to 256
parameter prefill_lvl  = 0;  // RANGE 0 to 255
parameter f_sync_type  = 2;  // RANGE 0 to 3
parameter reg_stat     = 1;  // RANGE 0 to 1
parameter tst_mode     = 0;  // RANGE 0 to 1
parameter verif_en     = 2;  // RANGE 0 to 2
parameter r_sync_type  = 2;  // RANGE 0 to 3
parameter clk_d_faster = 1;  // RANGE 0 to 15
parameter reg_in_prog  = 1;  // RANGE 0 to 1



input inst_clk_s;
input inst_rst_s_n;
input inst_init_s_n;
input inst_clr_s;
input inst_send_s;
input [width-1 : 0] inst_data_s;
output clr_sync_s_inst;
output clr_in_prog_s_inst;
output clr_cmplt_s_inst;

input inst_clk_d;
input inst_rst_d_n;
input inst_init_d_n;
input inst_clr_d;
input inst_prefill_d;
output clr_in_prog_d_inst;
output clr_sync_d_inst;
output clr_cmplt_d_inst;
output data_avail_d_inst;
output [width-1 : 0] data_d_inst;
output prefilling_d_inst;

input inst_test;


// Instance of DW_stream_sync
```

```
DW_stream_sync #(width, depth, prefill_lvl, f_sync_type, reg_stat, tst_mode, verif_en,
r_sync_type, clk_d_faster, reg_in_prog) U1 (
            .clk_s(inst_clk_s),
            .rst_s_n(inst_rst_s_n),
            .init_s_n(inst_init_s_n),
            .clr_s(inst_clr_s),
            .send_s(inst_send_s),
            .data_s(inst_data_s),
            .clr_sync_s(clr_sync_s_inst),
            .clr_in_prog_s(clr_in_prog_s_inst),
            .clr_cmplt_s(clr_cmplt_s_inst),

            .clk_d(inst_clk_d),
            .rst_d_n(inst_rst_d_n),
            .init_d_n(inst_init_d_n),
            .clr_d(inst_clr_d),
            .prefill_d(inst_prefill_d),
            .clr_in_prog_d(clr_in_prog_d_inst),
            .clr_sync_d(clr_sync_d_inst),
            .clr_cmplt_d(clr_cmplt_d_inst),
            .data_avail_d(data_avail_d_inst),
            .data_d(data_d_inst),
            .prefilling_d(prefilling_d_inst),

            .test(inst_test)
            );

endmodule
```

Revision History

For notes about this release, see the *DesignWare Building Block IP Release Notes*.

For lists of both known and fixed issues for this component, refer to the STAR report.

For a version of this datasheet with visible change bars, click here.

Date	Release	Updates
July 2020	DWBB_201912.5	■ Adjusted the material in "Simulation Methodology" on page 8 ■ Adjusted content and title of "Suppressing Warning Messages During Verilog Simulation" on page 8 and added the DW_SUPPRESS_WARN macro
October 2019	DWBB_201903.5	■ Added the "Disabling Clock Monitor Messages" section ■ Added this Revision History table and the document links on this page

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at https://www.synopsys.com/company/legal/trademarks-brands.html.

All other product or company names may be trademarks of their respective owners.

Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
www.synopsys.com